Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н. Э. Баумана (национальный исследовательский университет)»
(МГТУ им. Н. Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

ОТЧЁТ

По лабораторной работе №9

По курсу: «Функциональное и логическое программирование»

Студент: Керимов А. Ш.

Группа: ИУ7-64Б

Преподаватели: Толпинская Н. Б.,
Строганов Ю. В.

Москва

2020

Практическая часть

Задание 1. Написать предикат `set-equal`, который возвращает `t`, если два его множество-аргумента содержат одни и те же элементы, порядок которых не имеет значения.

```
(defun set-equal (a b)
  (null (set-exclusive-or a b)))
```

```
(defun fset-equal (a b)
  (if (= (list-length a) (list-length b))
      (reduce
        #'(lambda (acc cur)
            (if acc (member cur b) nil))
        a
        :initial-value t)
      nil))
```

```
(defun rset-equal (a b)
  (defun rset-equal-process (a)
    (cond ((null a) t)
          ((member (car a) b) (rset-equal-process (cdr a)))
          (t nil)))
  (if (= (list-length a) (list-length b))
      (rset-equal-process a)
      nil))
```

Примеры работы:

a	b	(fset-equal a b)	(rset-equal a b)
(1 2 3)	(3 2 1)	(3 2 1)	t
(1 2 3)	(3 2)	nil	nil
(1 2 3)	(3 2 0)	nil	nil

Задание 2. Напишите необходимые функции, которые обрабатывают таблицу из точечных пар: `(страна . столица)`, и возвращают по стране — столицу, а по столице — страну.

```
(defun get-by-template (getter setter dict value)
  (reduce
    #'(lambda (acc cur)
        (cond (acc acc)
              ((equal value (funcall getter cur)) (funcall setter cur))
              (t nil)))
    dict
   :initial-value nil))

(defun get-capital-by-country (dict country)
  (get-by-template #'car #'cdr dict country))

(defun get-country-by-capital (dict capital)
  (get-by-template #'cdr #'car dict capital))
```

```
(defun get-by-template (getter setter dict value)
  (cond ((null dict) nil)
        ((equal (funcall getter dict) value) (funcall setter dict))
        (t (get-by-template getter setter (cdr dict) value))))

(defun get-capital-by-country (dict country)
  (get-by-template #'caar #'cdar dict country))
```

```
(defun get-country-by-capital (dict capital)
  (get-by-template #'cdar #'caar dict capital))
```

Пример работы:

dict	`((a . b) (b . c) (c . d))`
value	`c`
(get-capital-by-country dict value)	`d`
(get-country-by-capital dict value)	`b`

Задание 3. Напишите функцию, которая умножает на заданное число-аргумент все числа из заданного списка-аргумента, когда а) все элементы списка — числа, б) элементы списка — любые объекты.

```
(defun mult-a (lst number)
  (mapcar
    #'(lambda (item)
        (* item number))
    lst))

(defun mult-b (lst number)
  (mapcar
    #'(lambda (item)
        (if (numberp item)
            (* item number)
            item))
    lst))
```

```
(defun mult-a (lst number)
  (defun mult-a-process (lst acc)
    (if lst
        (mult-a-process
          (cdr lst)
          (append acc
            (list (* (car lst) number))))
        acc))
  (mult-a-process lst nil))

(defun mult-b (lst number)
  (defun mult-b-process (lst acc)
    (if lst
        (mult-b-process
          (cdr lst)
          (append acc
            (if (numberp (car lst))
                (list (* number (car lst)))
                (list (car lst)))))
        acc))
  (mult-b-process lst nil))
```

Пример работы:

- `(mult-a '(1 2 3 4) 2) -> (2 4 6 8)`
- `(mult-b '(1 2 a 4) 2) -> (2 4 A 8)`
- `(mult-a '() 3) -> ()`

- `(mult-b '() 3) -> ()`

Задание 4. Напишите функцию, которая уменьшает на 10 все числа из списка аргумента этой функции.

```
(defun minus (lst number)
  (mapcar
    #'(lambda (item)
        (- item number))
    lst))

(defun minus-ten (lst)
  (minus lst 10))
```

```
(defun minus (lst number)
  (defun minus-process (lst acc)
    (if lst
        (minus-process
          (cdr lst)
          (append acc
            (list (- (car lst) number))))
        acc))
  (minus-process lst nil))

(defun minus-10 (lst)
  (minus lst 10))
```

Пример работы:

- `(minus-10 '(5 10 15)) -> (-5 0 5)`
- `(minus-10 '()) -> ()`

Задание 5. Написать функцию, которая возвращает первый аргумент списка-аргумента, который сам является непустым списком.

```
(defun get-first-sublist (lst)
  (reduce
    #'(lambda (acc cur)
        (cond (acc acc)
              ((listp cur) cur)
              (t nil)))
    lst
    :initial-value nil))
```

```
(defun not-empty-list (lst)
  (and (listp lst) lst))

(defun get-first-sublist (lst)
  (cond ((null lst) nil)
        ((not-empty-list (car lst)) (car lst))
        (t (get-first-sublist (cdr lst)))))
```

Пример работы:

- `(get-first-sublist '(1 2 3 (4 5) 6 7 (8 9 (10)))) -> (4 5)`
- `(get-first-sublist '(1 2 3 4)) -> nil`

Задание 6. Написать функцию, которая выбирает из заданного списка только те числа, которые больше 1 и меньше 10. (Вариант: между двумя заданными границами)

```
(defun between (lst a b)
  (reduce
    #'(lambda (acc cur)
        (if (and (< a cur) (< cur b))
            (append acc (list cur))
            acc))
    lst
    :initial-value nil))

(defun between-1-10 (lst)
  (between lst 1 10))
```

```
(defun between (lst a b)
  (defun is-between (number)
    (and (< a number) (< number b)))
  (defun between-process (lst acc)
    (if lst
        (between-process
          (cdr lst)
          (if (is-between (car lst))
              (append acc (list (car lst)))
              acc))
        acc))
  (between-process lst nil))

(defun between-1-10 (lst)
  (between lst 1 10))
```

Пример работы:

- `(between-1-10 '(-1 0 1 5 9 10 11 2)) -> (5 9 2)`
- `(between-1-10 '()) -> ()`

Задание 7. Написать функцию, вычисляющую декартово произведение двух своих списков-аргументов. (Напомним, что $A \times B$ — это множество всевозможных пар $(a\ b)$, где $a \in A$, $b \in B$)

```
(defun set-mult (a b)
  (apply
    #'append
        (mapcar
          #'(lambda (item-a)
              (mapcar
                #'(lambda (item-b)
                    (list item-a item-b))
                b))
          a)))
```

```
(defun set-mult (a b)
  (defun set-mul-process (x y)
    (cond ((null x) nil)
          ((null y) (set-mul-process (cdr x) b))
          (t (cons (list (car x) (car y))
                   (set-mul-process x (cdr y))))))
  (set-mul-process a b))
```

Пример работы:

- `(set-mult '(1 2) '(3 4 5)) -> ((1 3) (1 4) (1 5) (2 3) (2 4) (2 5))`
- `(set-mult '(1 2 3) '()) -> ()`

Задание 8. Почему так реализовано `reduce`, в чем причина?

```
(reduce #'+ 0)   ; 0
(reduce #'+ ())  ; 0
```

Из математических соображений

$$\sum_{a \in \varnothing} a = 0, \quad \prod_{a \in \varnothing} a = 1. \tag{1}$$

Теоретическая часть

Способы организации повторных вычислений в Lisp

Для организации повторных вычислений в Lisp могут быть использованы функционалы и рекурсия.

Различные способы использования функционалов

В Lisp используются применяющие и отображающие функционалы, функционалы, являющиеся предикатами, функционалы, использующие предикаты в качестве функционального объекта.

Что такое рекурсия? Способы организации рекурсивных функций

Рекурсия — это ссылка на определяемый объект во время его определения.

Существуют типы рекурсивных функций: хвостовая, дополняемая, множественная, взаимная рекурсия и рекурсия более высокого порядка.

При организации рекурсии можно использовать как функции с именем, так и локально определенные с помощью лямбда-выражений функции. Кроме этого, при организации рекурсии можно использовать функционалы или использовать рекурсивную функцию внутри функционала. При изучении рекурсии рекомендуется организовывать и отлаживать реализацию отдельных подзадач исходной задачи, обращая внимание на эффективность реализации и качество работы, а потом, при необходимости, встраивать эти функции в более крупные, возможно в виде лямбда-выражений.

Способы повышения эффективности реализации рекурсии

- **Хвостовая рекурсия**. В целях повышения эффективности рекурсивных функций рекомендуется формировать результат не на выходе из рекурсии, а на входе в рекурсию, все действия выполняя до ухода на следующий шаг рекурсии.

  Преобразование не хвостовой рекурсии в хвостовую, возможно путем использования дополнительных параметров. В этом случае необходимо использовать функцию-оболочку для запуска рекурсивной функции с начальными значениями дополнительных параметров.

- **Дополняемая рекурсия**. При обращении к рекурсивной функции используется дополнительная функция не в аргументе вызова, а вне его.

- **Выделяют группу функций множественной рекурсии**. На одной ветке происходит сразу несколько рекурсивных вызовов. Количество условий выхода также может зависеть от задачи.